Information Security Lab - AS21

Reductions

Handout: Oct 22, 2021 12:00 AM

Due: Nov 8, 2021 3:00 PM

SquareDH to DDH

Open Task

Reduction of SquareDH to DDH (25 points)

Generic Group Model

In this task, you will work in the **Generic Group Model** (GGM). In the GGM, it is assumed that there is a cyclic group $\mathbb{G}$ of prime order p (where $p \in \Theta(2^\lambda)$) to which you only have *oracle access*. I.e., each element $g \in \mathbb{G}$ is given to you by a random binary string, a *handle*, which represents this element but does not leak any information about its exponent. Given two handles of group elements g, h in the GGM, you need to ask an oracle to receive a handle of the product $g \cdot h$ or of a power g^e for $e \in \mathbb{Z}$.

For this task, we have already implemented the GGM and will provide you with an interface which encapsulates all necessary functionalities:

```
public interface IRandomGroupElement extends IBasicGroupElement<IRandomGroupEle
    BigInteger getGroupOrder();
    IRandomGroupElement multiply(IRandomGroupElement otherElement);
    IRandomGroupElement power(BigInteger exponent);
    IRandomGroupElement power(IRandomVariable exponent);
    IRandomGroupElement invert();
    IRandomGroupElement clone();
    boolean equals(IRandomGroupElement otherElement);
}
```

Each group element -- which you are given in this task -- will be of the type `IRandomGroupElement`. Use the methods of this interface to perform generic group operations.

Decisional Diffie-Hellman Assumption

In this task, your job is to show that if Square Diffie-Hellman is hard in $\mathbb{G}$ then the Decisional Diffie-Hellman assumption is hard, too. For this end, you will write a generic reduction which can decide SqDH-challenges when given access to an adversary which can decide DDH-challenges.

Let us first define the DDH assumption (as in the lecture): Let $g \in \mathbb{G}$ be a fixed generator of the cyclic group $\mathbb{G}$ of order p. We define the distribution $\mathcal{D}_{real}$ by:

$\mathcal{D}_{real}$:
Draw $x, y \leftarrow \mathbb{Z}_p$ uniformly and independently at random.
Output $(g, g^x, g^y, g^{x \cdot y})$.

And the distribution $\mathcal{D}_{random}$ by:

$\mathcal{D}_{random}$:
Draw $x, y, z \leftarrow \mathbb{Z}_p$ uniformly and independently at random.
Output (g, g^x, g^y, g^z).

The challenge of the DDH-assumption is now to distinguish samples of $\mathcal{D}_{real}$ from samples of $\mathcal{D}_{random}$. More precisely, the associated security game works as follows:

1. The DDH-challenger $\mathcal{C}$ draws a random boolean `isReal` $\leftarrow$ { `true`, `false` }.
2. If `isReal` is true, $\mathcal{C}$ draws $(g, g^x, g^y, g^z) \leftarrow \mathcal{D}_{real}$. Otherwise, it draws $(g, g^x, g^y, g^z) \leftarrow \mathcal{D}_{random}$. (So, $z = x \cdot y$ does hold iff `isReal` is true.)
3. $\mathcal{C}$ starts a DDH-adversary $\mathcal{A}$ and hands him over the DDH-challenge (g, g^x, g^y, g^z) when $\mathcal{A}$ asks for it.
4. $\mathcal{A}$ performs some generic computations on (g, g^x, g^y, g^z) and comes up with a boolean value `solution` which it sends back to $\mathcal{C}$.
5. $\mathcal{C}$ checks if `solution == isReal`. If `solution == isReal` then $\mathcal{A}$ wins. Otherwise, $\mathcal{A}$ loses.

The **advantage** of $\mathcal{A}$ in this security game is defined by
$Adv_{DDH}(\mathcal{A}) := 2 \cdot Pr[\mathcal{A} \text{ wins}] - 1.$
The **DDH-assumption** states that for each ppt adversary $\mathcal{A}$ its advantage in winning the above is negligible (in the security parameter).

So, the decisional Diffie-Hellman assumption basically states that it is hard to distinguish the product in the exponent of $g^{x \cdot y}$ of the exponents of two group elments g^x, g^y from uniform randomness.

To encapsulate DDH challenges in this task, we created a container class which you should use:

```
public class DDH_Challenge<IRandomGroupElement> {
    public final IRandomGroupElement generator;
    public final IRandomGroupElement x;
    public final IRandomGroupElement y;
    public final IRandomGroupElement z;
}
```

Decisional Square Diffie-Hellman Assumption

The Square Diffie-Hellman Assumption is defined analogously:

Let $g \in \mathbb{G}$ be a fixed generator of the cyclic group $\mathbb{G}$ of order p. We define the distribution $\mathcal{S}_{real}$ by:

$\mathcal{S}_{real}$:
Draw $a \leftarrow \mathbb{Z}_p$ uniformly and independently at random.
Output (g, g^a, g^{a^2}).

And the distribution $\mathcal{S}_{random}$ by:

$\mathcal{S}_{random}$:
Draw $a, b \leftarrow \mathbb{Z}_p$ uniformly and independently at random.
Output (g, g^a, g^b).

The associated security game is given by:

1. The SqDH-challenger $\mathcal{C}$ draws a random boolean `isReal` $\leftarrow$ { `true`, `false` }.
2. If `isReal` is true, $\mathcal{C}$ draws $(g, g^a, g^b) \leftarrow \mathcal{S}_{real}$. Otherwise, it draws $(g, g^a, g^b) \leftarrow \mathcal{S}_{random}$. (So, $b = a^2$ does hold iff `isReal` is true.)
3. $\mathcal{C}$ starts a SqDH-adversary $\mathcal{A}$ and hands him over the SqDH-challenge (g, g^a, g^b) when $\mathcal{A}$ asks for it.
4. $\mathcal{A}$ performs some generic computations on (g, g^a, g^b) and comes up with a boolean value `solution` which it sends back to $\mathcal{C}$.
5. $\mathcal{C}$ checks if `solution == isReal`. If `solution == isReal` then $\mathcal{A}$ wins. Otherwise, $\mathcal{A}$ loses.

The **advantage** of $\mathcal{A}$ in this security game is defined by
$Adv_{SqDH}(\mathcal{A}) := 2 \cdot Pr[\mathcal{A} \text{ wins}] - 1.$
The **SqDH-assumption** states that for each ppt adversary $\mathcal{A}$ its advantage in winning the above is negligible (in the security parameter).

So, the decisional square Diffie-Hellman assumption basically states that it is hard to distinguish the square in the exponent of g^{x^2} of the exponent of a group elment g^x from uniform randomness.

To encapsulate SqDH challenges in this task, we created a container class which you should use:

```
public class SquareDH_Challenge<IRandomGroupElement> {
    public final IRandomGroupElement generator;
    public final IRandomGroupElement a;
    public final IRandomGroupElement b;
}
```

The Reduction

Your job is to finish the implementation of the class `SquareDH_DDH_Reduction`:

```
public class SquareDH_DDH_Reduction extends A_SquareDH_DDH_Reduction<IRandomGrou

    public Boolean run(I_SquareDH_Challenger<IRandomGroupElement, IRandomVariable
        //...
    }

    public DDH_Challenge<IRandomGroupElement> getChallenge() {
        //...
```

```
    }
}
```

`SquareDH_DDH_Reduction` reduces the problem of the SqDH assumption to the problem of the DDH assumption. For this end, your reduction needs to decide a `SquareDH_Challenge`. To get this challenge, you need to ask the `I_SquareDH_Challenger<IRandomGroupElement, IRandomVariable> challenger` for a challenge by calling.

```
SquareDH_Challenge<IRandomGroupElement> challenge = challenger.getChallenge();
```

When you come up with a boolean `solution` you need to `return` it in the method `run`.

For deciding `challenge`, the class `SquareDH_DDH_Reduction` has a field `I_DDH_Adversary<IRandomGroupElement, IRandomVariable> adversary` which contains a perfect DDH Adversary:

```
public interface I_DDH_Adversary<IRandomGroupElement, IRandomVariable> {
     Boolean run(I_DDH_Challenger<IRandomGroupElement,IRandomVariable> challenge
}
```

When calling `adversary.run(this)`, `adversary` will try to ask for a DDH challenge from the reduction. For this end, you need to implement the method `getChallenge()` and return a `DDH_Challenge ddh_challenge` there.

When `adversary` can succesfully ask `ddh_challenge` it will try to decide `ddh_challenge`. This means, if `ddh_challenge` contains the group elements (g, g^x, g^y, g^z), `adversary` will return -- in an overwhelming number of cases -- `true` if $z = x \cdot y$ and `false` otherwise.

Now, given `adversary`, your task is to decide `SquareDH_Challenge challenge`. `challenge` will consist of three group elements `generator, a, b` which can be written as g, g^a, g^b. You need to return `true` if $b = a^2$ and `false` otherwise.

To be able to decide `challenge`, you need to make use of `adversary`. For this end, you need to transform the SqDH-Challenge `challenge` to a DDH challenge `ddh_challenge` which is a real DDH sample if and only if `challenge` is a real SqDH sample. You can then ask `adversary` if `ddh_challenge` is a real DDH sample.

Randomness

Note that `adversary` is only guaranteed to have an overwhelming advantage when the `ddh_challenge` you provide in `getChallenge()` is distributed according to $\mathcal{D}_{real}$ or $\mathcal{D}_{random}$.

This implies that in each DDH challenge (g, g^x, g^y, g^z) which you give to `adversary` the exponents x and y must be drawn uniformly and independently at random. To ensure that this holds we provide you with a fresh random variable which you can use to transform the SqDH challenge `challenge` to a correctly distributed DDH challenge.

You can get this random variable by calling:

```
IRandomVariable R = challenger.getRandomVariable();
```

You can only get **one** random variable, so do not call `getRandomVariable()` twice or more!

The random variable `R` will be drawn uniformly at random and independent of the exponents of the SqDH-challenge `challenge` which you received by `challenger`. `R` is of type `IRandomVariable`

```
public interface IRandomVariable {
    IRandomVariable add(IRandomVariable otherElement);
    IRandomVariable add(BigInteger number);
    IRandomVariable multiply(IRandomVariable otherElement);
    IRandomVariable multiply(BigInteger number);
    boolean isZero();
    boolean isConstant();
    IRandomVariable subtract(IRandomVariable otherElement);
    IRandomVariable subtract(BigInteger number);
    IRandomVariable negate();
}
```

and can be used just like an object of type `java.math.BigInteger` in most cases. To generate a correctly distributed DDH challenge use exactly the random variable `R` and the elements of the SqDH-Challenge `challenge`. Do **not** use any other sources of randomness like `java.util.Random` or `java.security.SecureRandom`!

Important Note: To make sure that the DDH challenge (g, g^x, g^y, g^z) you provide to `adversary` is distributed correctly you need to choose the elements g, g^x, g^y in a very special way:

- The generator g of the DDH challenge you give to `adversary` must be exactly the generator g of the SqDH-challenge which you recevied.
- The exponent x of g^x must be a where a is the exponent of the element g^a of the SqDH-Challenge (g, g^a, g^b) which you recevied.
- The exponent y of g^y must be $a \cdot R$ or $a + R$ where a is the exponent of the element g^a of the SqDH-Challenge (g, g^a, g^b) which you recevied and R is the random variable you received by the challenger.

Concretely, the DDH-challenge which you give to `adversary` must be created either as:

```
SquareDH_Challenge<IRandomGroupElement> challenge = challenger.getChallenge();
IRandomVariable R = challenger.getRandomVariable();
IRandomGroupElement gA = challenge.a;
```

```
IRandomGroupElement generator = challenge.generator;
IRandomGroupElement gX = gA;
IRandomGroupElement gY = gA.power(R);
IRandomGroupElement gZ = ...;

DDH_Challenge<IRandomGroupElement> ddh_challenge =
    new DDH_Challenge<IRandomGroupElement>(generator, gX, gY, gZ);
```

Or as:

```
SquareDH_Challenge<IRandomGroupElement> challenge = challenger.getChallenge();
IRandomVariable R = challenger.getRandomVariable();
IRandomGroupElement gA = challenge.a;

IRandomGroupElement generator = challenge.generator;
IRandomGroupElement gX = gA;
IRandomGroupElement gY = gA.multiply(generator.power(R));
IRandomGroupElement gZ = ...;

DDH_Challenge<IRandomGroupElement> ddh_challenge =
    new DDH_Challenge<IRandomGroupElement>(generator, gX, gY, gZ);
```

If the DDH-challenge you supply does not comply with the above rules, `adversary` is not guaranteed anymore to have a non-negligible advantage in deciding `ddh_challenge`. In this case, the behaviour of `adversary` will be erroneous and the boolean which it will return will be of no help for your reduction.

Tightness

In this task, we are interested in **tight** reductions. This means, your reduction may call `adversary.run` at most once.

Reductions which call `adversary.run` twice or more during a run will only receive **partial** points!

Constructors

Do **under no circumstances** change or remove the constructor of `SquareDH_DDH_Reduction` which we pre-implemented. The TestRunner needs this empty constructor to test your solution. If this constructor does not exist or work, then the TestRunner can not test your solution and you will receive 0 points.

Testing Your Implementation

To test your implementation, you can use the Run- and Test-Button of the Code-Expert GUI. When you do this, the TestRunner will try to compile your reduction and play the Square-DH security

game which we described above several hundred times with it to estimate the advantage of your reduction.

Scores and Points

If the measured advantage is high enough and your reduction is tight, then you should receive full points (25 of 25).

If your reduction is not tight, you will only receive partial points. If your reduction does not follow the rules which we explained here it might have a negligible advantage and will get zero points.

After each run, the TestRunner will tell you how many points your solution got in the *preliminary* tests.

Important Note: The tests which we run in Code Expert are only **preliminary**. After the submission deadline, we will run more exhaustive tests on your solution and review it manually.

Therefore, a solution which is only partially correct may receive full points on Code Expert in the preliminary tests but will get only partial points, eventually. Therefore, make sure that your reductions are correct in the formally theoretic sense of cryptographic reductions!

Time and Memory Restrictions

The resources the TestRunner can use to test your solution are limited. We expect your solution to use less than 10 seconds of CPU time and a restriced space of memory when run several hundred times.

Solutions which run into TimeOut- or OutOfMemoryExceptions will be rejected by us and receive 0 points.

Cheater Warning

The purpose of this task is to algorithmically reduce the decisional SquareDH problem to the decisional DH problem.

Any solution which tries to solve the decisional SqareDH problem by cryptoanalytical algorithms or by "tricking" the testing environment is considered to be a cheating attempt and will receive zero points.

Licensing

The TestRunner for this task uses the rings package of Stanislav Poslavsky which is licensed under Apache License, Version 2.0 http://www.apache.org/licenses/LICENSE-2.0.txt.

[POS19] Stanislav Poslavsky, Rings: An efficient Java/Scala library for polynomial rings, Computer Physics Communications, Volume 235, 2019, Pages 400-413, doi:10.1016/j.cpc.2018.09.005